

L4Re Boot Camp

Microkernel-based Operating Systems—Exercise

Barkhausen Institut, Dresden (Germany)

Carsten Weinhold, Viktor Reusch

WiSe 26/27

Agenda



- first contact with a microkernel OS
- talk about system booting
- getting to know QEMU
- compile L4Re microkernel (“Fiasco”)
- compile minimal system environment
- the usual “Hello World”
- look at source and config, play with it



- developing your own kernel usually requires a dedicated machine
- we will use a virtual machine
- QEMU is open-source, provides a virtual machine by binary translation
- it emulates a complete x86 PC
- ... many other system architectures, too
- our QEMU will boot from an ISO image

Booting



- Basic Input Output System
- fixed entry point after “power on” and “reset”
- initializes the CPU in 16-bit real-mode
- detects, checks, and initializes platform hardware (RAM, PCI(e), SATA, ...)
- finds the boot device



- Extensible Firmware Interface
 - more modern
 - plug-ins for new hardware
- no legacy PC-AT boot (no A20 gate)
- built-in boot manager
 - more than four partitions, no 2TB limit
 - boot from peripherals (USB)

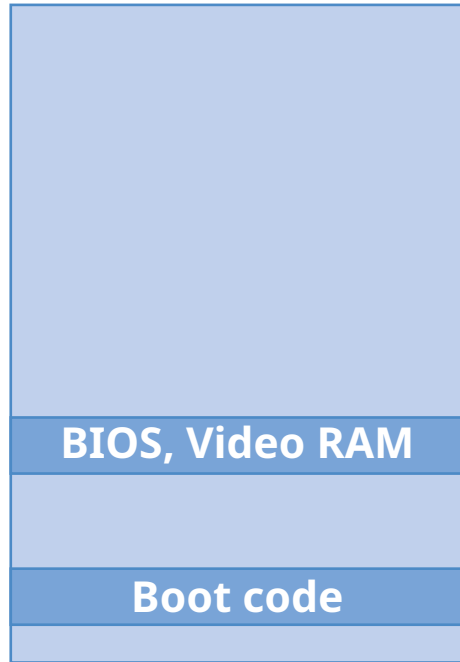
Boot Sector



- first sector on boot disk
- 512 bytes
- contains first boot loader stage and partition table
- BIOS loads code into RAM and executes it
- problem: How to find and boot an OS in 512 bytes?

BIOS

Memory Layout



PHYSICAL MEMORY

BIOS

- popular boot loader
- used by most (all?) Linux distributions
- we consider the x86 the legacy boot mode (non-EFI)
- GRUB legacy (v0.9.x) uses a two-stage-approach
 - first stage fits in one sector
 - has hard-wired sectors of second stage files
 - second stage can read common file systems
- GRUB2 also supports legacy mode, but introduces names for the two stages:
 - `boot.img` stored in boot sector
 - `core.img` stored in blocks following boot sector

Boot Loader

BIOS



- second stage loads a config file to present a boot menu
 - GRUB legacy: `menu.lst`
 - GRUB2: `grub.cfg`
- from there, you can load your kernel
- supports loading multiple modules
- files can also be retrieved from network

Boot Loader

BIOS

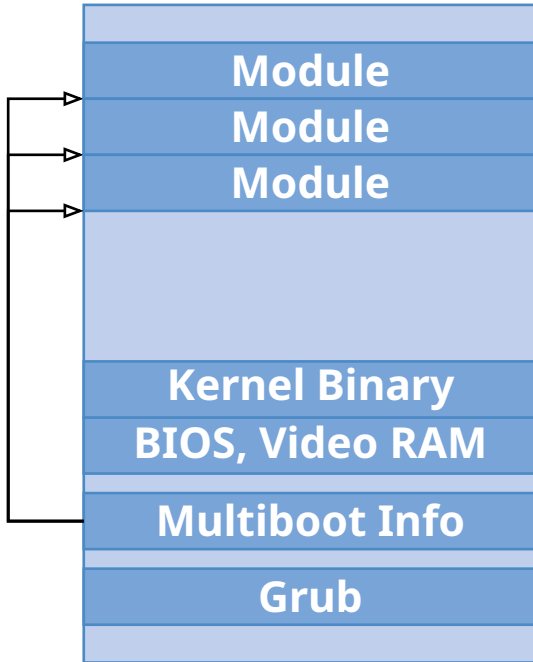


- switches CPU to 32-bit protected mode
- loads and interprets the „kernel“ binary
- loads additional modules into memory
- sets up multiboot info structure
- starts the kernel

Boot Loader

BIOS

Memory Layout



PHYSICAL MEMORY

Boot Loader

BIOS

- our modules are ELF files: executable and linkable format
- modules contain multiple sections
 - code
 - data
 - BSS
 - ...
- bootstrap interprets the ELF modules
- copies sections to final location in physical memory

Bootstrap

Boot Loader

BIOS

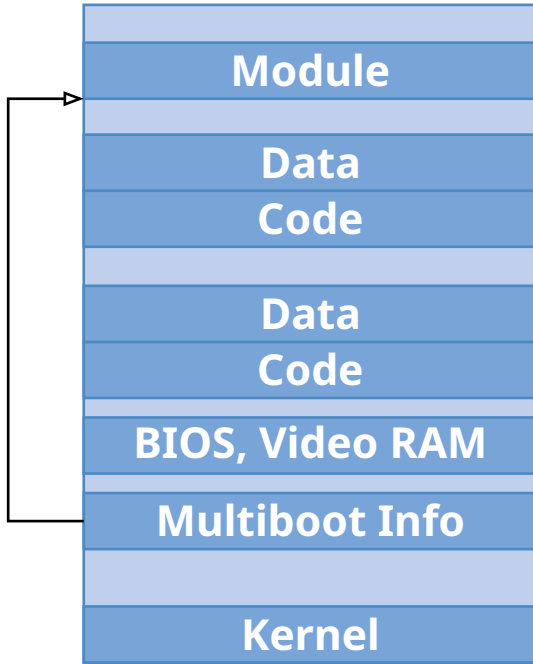
- actual L4Re kernel is the first of the modules
- must know about the other modules
- bootstrap sets up a kernel info page
 - contains entry point + stack pointer of sigma0 and moe
- passes control to the kernel

Bootstrap

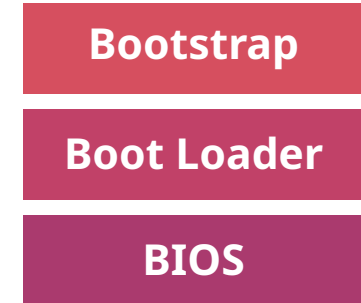
Boot Loader

BIOS

Memory Layout



PHYSICAL MEMORY





- initial kernel code
- basic CPU setup
 - detects CPU features
 - sets up various CPU tables
- sets up basic page table
- enables virtual memory mode
- jumps to the actual kernel code

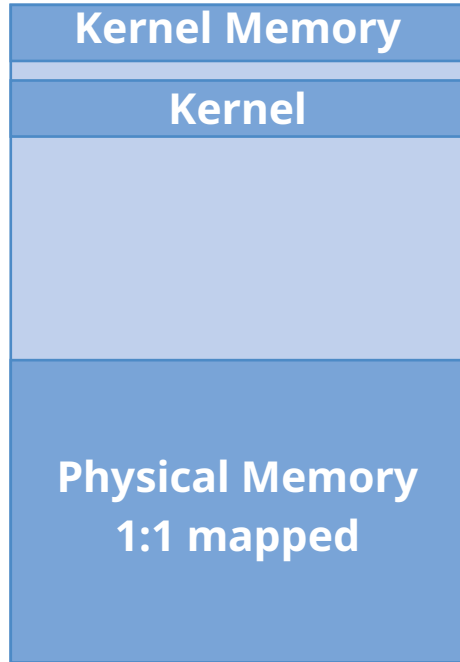
Kernel Loader

Bootstrap

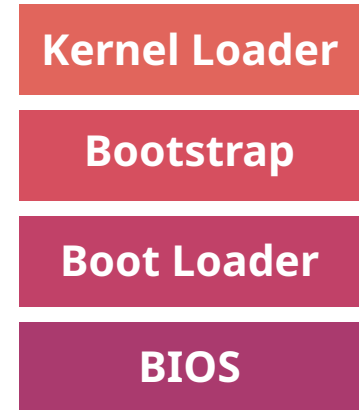
Boot Loader

BIOS

Memory Layout



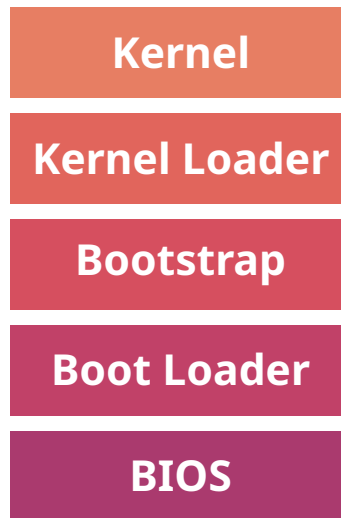
VIRTUAL MEMORY



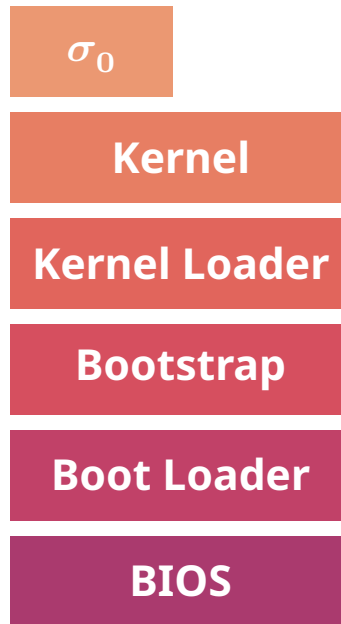
L4Re Microkernel (Fiasco)



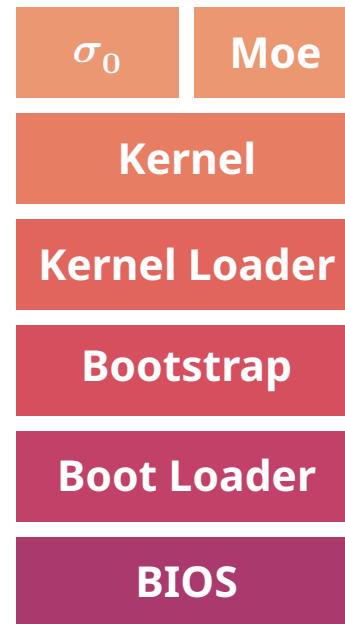
- sets up kernel structures
- sets up scheduling timer
- starts first pager (sigma0)
- starts first task (root task “Moe”)
- starts scheduling
- scheduler hands control to userland for the first time



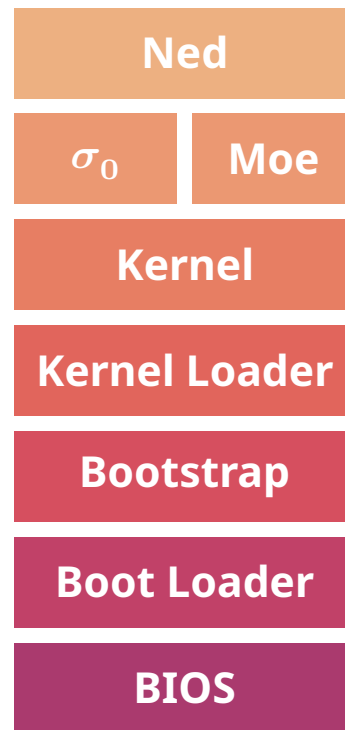
- is first pager in the system
- initially receives a 1:1 map-ping of physical memory
- ... and other platform-level resources (I/O ports)
- sigma0 is the root of the pager hierarchy
- pager for moe



- manages initial resources
 - namespace
 - memory
 - VESA framebuffer
- provides logging facility
- mini-filesystem for read-only access to boot-modules



- script-driven loader for further programs
 - loads programs into memory and configures their initial capabilities
 - startup-scripts written in Lua
- software can be loaded by retrieving binaries from disk or network
- ned injects common service code into every task



Practical Part

- Website: <https://tu-dresden.de/ing/informatik/sya/professur-fuer-betriebssysteme/studium/vorlesungen/mos>
- Mailinglist: mos2026@os.inf.tu-dresden.de
Subscribe at <https://os.inf.tu-dresden.de/mailman/listinfo/mos2026>
- Repository: <https://os.inf.tu-dresden.de/repo/gitbox/teaching/moslab-code.git>
- Matrix for discussing problems and solutions: <https://matrix.to/#/#mos-2026:os.inf.tu-dresden.de>

Requirements



- Basic OS knowledge: threads, processes, virtual memory, drivers, ...
- Experience with Linux: editor, shell, development (make, gcc)
- Experience with C++

Required Software

Option 0: Manual

- An x86-64/amd64 system
- A Linux system (preferably Debian/Ubuntu)
- qemu, perl
- build-essentials: gcc, g++, make, gawk, flex, bison
- gcc-multilib, g++-multilib
- device-tree-compiler
- git
- dialog, libncurses-dev
- grub2-common, xorriso, genisoimage



Required Software

Option 1: Podman/Docker

- Podman/Docker
- qemu, make
- Use the `container.sh` script
- qemu VM (likely) should run outside



Required Software

Option 2: Nix

- nix
- Use the `nix.sh` script

Required Software

Option 3: x86_64 VM

- A hypervisor/VMM e.g. VirtualBox
- We provide a ready-to-use VirtualBox image:
https://os.inf.tu-dresden.de/Studium/Bs/WS2025/Uebungen/VM_Image_BuS_2025.ova
- User login is `studi:studi`
- Root login is `root:bsvm`

Required Software

Option 4: Arm VM

- Get yourself an arm64 Debian VM:
<https://datashare.tu-dresden.de/s/MjGsgXxidsEFdQH>
- User login is `studi:studi`
- Root login is `root:studi`
- Additionally install: `g++-x86-64-linux-gnu`, `gdb-multiarch`
- For `grub-mkrescue` you need to run `grub.sh` and add the resulting `grub/` directory after the `-d` option
- Use `gdb-multiarch` for debugging

Required Software

Option 4: Arm VM

- Get yourself an arm64 Debian VM:
<https://datashare.tu-dresden.de/s/MjGsgXxidsEFdQH>
- User login is `studi:studi`
- Root login is `root:studi`
- Additionally install: `g++-x86-64-linux-gnu`, `gdb-multiarch`
- For `grub-mkrescue` you need to run `grub.sh` and add the resulting `grub/` directory after the `-d` option
- Use `gdb-multiarch` for debugging
- Alternatively, the Nix flake should work out of the box on `aarch64-linux`.



- clone the source code repository via
`git clone https://os.inf.tu-dresden.de/repo/gitbox/teaching/moslab-code.git`
- cd into the working directory and have a look around

The Repository



<code>doc/source/</code>	documentation (open <code>index.html</code> in a browser)
<code>obj/l4/amd64/</code>	build directory
<code>src/fiasco/</code>	Fiasco kernel sources
<code>src/l4/pkg/</code>	sources for all userspace packages
<code>src/l4/conf/</code>	boot configuration

Compiling The System (1)



- Build the L4Re userland:

```
[mos] $ cd src/l4
```

```
[mos/src/l4] $ make B=../../obj/l4/amd64
```

```
[mos/src/l4] $ cd ../../obj/l4/amd64
```

```
[mos/obj/l4/amd64] $ make -j <NUMBER-OF_CPUS>
```

- If desired, generate the `compile_commands.json` for *clangd*:

```
[mos/src/l4] make compile_commands.json O=../../obj/l4/amd64
```

Compiling The System (2)



- Build the L4Re microkernel ("fiasco"):

```
[mos] $ cd src/fiasco
```

```
[mos/src/fiasco] $ make B=../../obj/fiasco/amd64
```

```
[mos/src/fiasco] $ cd ../../obj/fiasco/amd64
```

```
[mos/obj/fiasco/amd64] $ make -j <NUMBER-OF_CPUS>
```



- create a bootable ISO image
 - the `iso` subdirectory is for the ISO's content
 - run `grub-mkrescue --output=boot.iso iso`
- your ISO will contain a minimal grub installation
- launch QEMU with the resulting ISO:
`qemu-system-x86_64 -m 512 -cdrom boot.iso`



- copy some files to the ISO directory
 - fiasco from the Fiasco build directory:
obj/fiasco/amd64/
 - bootstrap from:
obj/l4/amd64/bin/amd64_gen/plain/
 - sigma0, moe, l4re and ned from:
obj/l4/amd64/bin/amd64_gen/l4f/

- edit iso/boot/grub/grub.cfg:

```
menuentry "Getting Started" {  
    multiboot2 /bootstrap -serial  
    module2 /fiasco  
    module2 /sigma0  
    module2 /moe moe  
    module2 /l4re l4re  
    module2 /ned ned  
}
```
- rebuild the ISO and run QEMU



- create the file `hello.lua` in the `iso` directory with this content:

```
local L4 = require("L4");  
L4.default_loader:start({}, "rom/hello");
```

- pass ned this new startup script
- add this line to `grub.cfg`:
`module2 /hello.lua hello.lua`
- pass `rom/hello.lua` as parameter to `moe`
- load the future `hello` module in `grub.cfg`

Assignment 1.1: Hello World



- create a directory for your hello-project in `src/l4/pkg`
- create a `Control` file with the following content:

```
requires: stdlibs libstdc++
```

- create a `Makefile` with the following content:

```
PKGDIR      ?= .  
L4DIR       ?= $(PKGDIR)/../..  
TARGET      = hello  
SRC_CC      = hello.cc  
REQUIRES_LIBS = libstdc++  
include $(L4DIR)/mk/prog.mk
```

- fill in `hello.cc` and re-compile `l4` and run in QEMU via the GRUB ISO

Assignment 1.2: Ackermann



- write a program that spawns six threads
 - you can use `std::thread` in our system
- each thread should calculate one value $A(3, 0..5)$ of the Ackermann function:

$$A(0, m) = m + 1$$

$$A(n, 0) = A(n - 1, 1)$$

$$A(n, m) = A(n - 1, A(n, m - 1))$$

Assignment 1.2: Ackermann



- write a program that spawns six threads
 - you can use `std::thread` in our system
- each thread should calculate one value $A(3, 0..5)$ of the Ackermann function:

$$A(0, m) = m + 1$$

$$A(n, 0) = A(n - 1, 1)$$

$$A(n, m) = A(n - 1, A(n, m - 1))$$

- You can form groups of up to three students and work together.
- We will discuss your solution next time we meet for this lab.

Get in Touch



Make sure you subscribe to the mailing list and join our Matrix room!
See slide 23 for the links.